**Техническое задание для ИИ: Автоматизированный бот обработки email**

**🎯 ОБЩАЯ ЗАДАЧА**

Создать Python-приложение, которое:

1. Подключается к Zoho Mail через IMAP
2. Находит отправленные письма за последние 2 недели
3. Для каждого отправленного письма ищет ответ от получателя
4. Анализирует найденные ответы через LM Studio API
5. Обновляет данные в Excel файле на основе анализа

**📋 ДЕТАЛЬНЫЕ ТРЕБОВАНИЯ**

**1. СТРУКТУРА ПРОЕКТА**

Создать следующую структуру файлов:

email\_bot/

├── main.py # Основной файл запуска

├── config/

│ ├── \_\_init\_\_.py

│ ├── settings.py # Конфигурация приложения

│ └── config.yaml # Файл настроек

├── src/

│ ├── \_\_init\_\_.py

│ ├── imap\_client.py # Работа с IMAP

│ ├── excel\_processor.py # Работа с Excel

│ ├── lm\_studio\_client.py # Работа с LM Studio API

│ └── utils.py # Вспомогательные функции

├── logs/ # Папка для логов (создается автоматически)

├── requirements.txt # Зависимости

├── .env.example # Пример файла окружения

└── README.md # Документация

**2. КОНФИГУРАЦИЯ (config/settings.py и config.yaml)**

**config.yaml должен содержать:**

imap:

host: "imappro.zoho.eu"

port: 993

timeout: 30

retry\_attempts: 3

folders:

sent: ["Sent Items", "Sent"]

inbox: ["INBOX"]

search:

days\_back: 14

max\_emails\_per\_batch: 100

lm\_studio:

timeout: 90

max\_tokens: 512

temperature: 0.0

retry\_attempts: 2

excel:

target\_columns:

- "Price usd"

- "Price usd casino"

- "Payment"

- "Language"

- "Comments"

mail\_column: "Mail"

backup: true

highlight\_changes: true

logging:

level: "INFO"

file: "bot.log"

format: "[%(asctime)s] [%(levelname)s] %(message)s"

date\_format: "%Y-%m-%d %H:%M:%S"

**settings.py должен загружать конфигурацию:**

import yaml

import os

from pathlib import Path

class Settings:

def \_\_init\_\_(self):

config\_path = Path(\_\_file\_\_).parent / "config.yaml"

with open(config\_path, 'r', encoding='utf-8') as f:

self.config = yaml.safe\_load(f)

def get(self, key\_path, default=None):

"""Получить значение по пути типа 'imap.host'"""

keys = key\_path.split('.')

value = self.config

for key in keys:

value = value.get(key, {})

return value if value != {} else default

settings = Settings()

**3. СИСТЕМА ЛОГИРОВАНИЯ**

Создать централизованную систему логирования в utils.py:

import logging

import os

from datetime import datetime

from pathlib import Path

def setup\_logging(log\_level="INFO", log\_file="bot.log"):

"""

Настройка логирования с записью в файл и вывод в консоль

"""

# Создание папки logs если не существует

log\_dir = Path("logs")

log\_dir.mkdir(exist\_ok=True)

# Настройка логгера

logger = logging.getLogger('EmailBot')

logger.setLevel(getattr(logging, log\_level.upper()))

logger.handlers.clear() # Очистка существующих handlers

# Форматтер

formatter = logging.Formatter(

'[%(asctime)s] [%(levelname)s] [%(name)s] %(message)s',

datefmt='%Y-%m-%d %H:%M:%S'

)

# Файловый handler

file\_handler = logging.FileHandler(

log\_dir / log\_file,

encoding='utf-8'

)

file\_handler.setLevel(logging.DEBUG)

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

# Консольный handler

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.INFO)

console\_handler.setFormatter(formatter)

logger.addHandler(console\_handler)

return logger

**4. IMAP КЛИЕНТ (src/imap\_client.py)**

Создать класс для работы с IMAP с обязательными методами:

**Требования к классу IMAPClient:**

1. **Контекстный менеджер** для автоматического закрытия соединений
2. **Повторные попытки** при сбоях соединения
3. **Правильная обработка кодировок** в заголовках писем
4. **Поиск писем по различным критериям**

class IMAPClient:

def \_\_init\_\_(self, host, port, email, password, logger):

"""Инициализация клиента"""

pass

def \_\_enter\_\_(self):

"""Контекстный менеджер - вход"""

pass

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

"""Контекстный менеджер - выход"""

pass

def connect(self):

"""Подключение к IMAP серверу с повторными попытками"""

pass

def get\_sent\_emails(self, days\_back=14):

"""

Получить отправленные письма за указанный период

Возвращает: List[Dict] с полями:

- to: email получателя

- message\_id: ID сообщения

- subject: тема письма

- normalized\_subject: нормализованная тема

- date: дата отправки

- body: тело письма

- references: цепочка References

"""

pass

def find\_reply(self, sent\_email\_data):

"""

Найти ответ на отправленное письмо

Стратегии поиска (по приоритету):

1. По In-Reply-To заголовку

2. По References цепочке

3. По FROM + SUBJECT + дате

4. По FROM + частичному SUBJECT

5. По FROM + дате (последнее письмо)

Возвращает: Dict с полями from, subject, body, date или None

"""

pass

**Особые требования к поиску:**

1. **Нормализация темы письма**: удалять "RE:", "FWD:", номера в скобках
2. **Безопасный поиск**: экранировать специальные символы в IMAP запросах
3. **Обработка кодировок**: корректно декодировать заголовки Subject, From
4. **Извлечение текста**: приоритет text/plain, fallback на text/html с удалением HTML тегов

**5. LM STUDIO КЛИЕНТ (src/lm\_studio\_client.py)**

Создать класс для работы с LM Studio API:

class LMStudioClient:

def \_\_init\_\_(self, api\_url, model\_name, logger, timeout=90):

"""Инициализация клиента"""

pass

def analyze\_email(self, email\_body, target\_fields, retries=2):

"""

Анализ письма через LM Studio

Параметры:

- email\_body: текст письма для анализа

- target\_fields: список полей для извлечения

- retries: количество повторных попыток

Возвращает: Dict с извлеченными данными или None при ошибке

"""

pass

def \_create\_prompt(self, email\_body, target\_fields):

"""

Создание промпта для LM Studio

ВАЖНО: Промпт должен быть на русском языке, четко структурирован

Требовать JSON ответ без дополнительных объяснений

"""

prompt = f"""

Проанализируй текст письма и извлеки следующую информацию в формате JSON.

Если информация отсутствует, оставь поле пустым ("").

Поля для извлечения:

{chr(10).join(f'- {field}' for field in target\_fields)}

Верни только JSON без дополнительных комментариев.

Текст письма:

{email\_body}

"""

return prompt

def \_parse\_response(self, response\_text, target\_fields):

"""

Парсинг ответа от LM Studio

ВАЖНО:

1. Найти JSON в тексте (может быть окружен другим текстом)

2. Удалить комментарии // из JSON

3. Валидировать структуру

4. Вернуть словарь с нужными полями

"""

pass

**6. EXCEL ПРОЦЕССОР (src/excel\_processor.py)**

Создать класс для работы с Excel файлами:

class ExcelProcessor:

def \_\_init\_\_(self, file\_path, logger, mail\_column="Mail", target\_columns=None):

"""Инициализация процессора"""

pass

def load\_data(self):

"""

Загрузка Excel файла

Проверка существования файла и необходимых колонок

"""

pass

def create\_email\_index(self):

"""

Создание индекса email -> [список индексов строк]

Нормализация email адресов (приведение к нижнему регистру, trim)

"""

pass

def update\_rows(self, email, extracted\_data):

"""

Обновление строк в DataFrame для указанного email

Отслеживание изменений для подсветки

"""

pass

def save\_with\_backup(self):

"""

Сохранение файла с созданием резервной копии

Подсветка измененных ячеек желтым цветом

"""

pass

def highlight\_changes(self):

"""

Подсветка измененных ячеек в Excel файле

Использовать openpyxl для форматирования

"""

pass

**7. ОСНОВНАЯ ЛОГИКА (main.py)**

Создать главный файл с четкой последовательностью действий:

def main():

"""

Основной процесс обработки:

1. Загрузка конфигурации и настройка логирования

2. Проверка переменных окружения

3. Инициализация клиентов (IMAP, LM Studio, Excel)

4. Получение отправленных писем

5. Для каждого письма:

- Поиск ответа

- Анализ через LM Studio

- Обновление Excel

6. Сохранение результатов

7. Вывод статистики

"""

pass

class ProcessingStats:

"""

Класс для сбора и отображения статистики обработки

"""

def \_\_init\_\_(self):

self.start\_time = time.time()

self.total\_sent = 0

self.replies\_found = 0

self.lm\_analysis\_success = 0

self.excel\_updates = 0

self.errors = {'imap': 0, 'lm\_studio': 0, 'excel': 0}

def log\_summary(self, logger):

"""Вывод детальной статистики"""

pass

**8. ОБРАБОТКА ОШИБОК И ПОВТОРНЫЕ ПОПЫТКИ**

Реализовать универсальный декоратор для повторных попыток:

import time

import random

from functools import wraps

def retry\_with\_backoff(max\_retries=3, base\_delay=1, backoff\_factor=2):

"""

Декоратор для повторных попыток с экспоненциальной задержкой

"""

def decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

for attempt in range(max\_retries):

try:

return func(\*args, \*\*kwargs)

except Exception as e:

if attempt == max\_retries - 1:

raise e

delay = base\_delay \* (backoff\_factor \*\* attempt) + random.uniform(0, 1)

time.sleep(delay)

# Логирование повторной попытки

if hasattr(args[0], 'logger'):

args[0].logger.warning(f"Retry {attempt + 1}/{max\_retries} for {func.\_\_name\_\_} after {delay:.2f}s: {e}")

return None

return wrapper

return decorator

**9. ВАЛИДАЦИЯ ДАННЫХ**

Создать функции валидации в utils.py:

import re

from email.utils import parseaddr

def validate\_email(email):

"""Проверка корректности email адреса"""

pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

return re.match(pattern, email.strip()) is not None

def validate\_excel\_structure(df, required\_columns):

"""

Проверка структуры Excel файла

- Наличие обязательных колонок

- Корректность email адресов

- Отсутствие дубликатов email

"""

missing\_columns = [col for col in required\_columns if col not in df.columns]

if missing\_columns:

raise ValueError(f"Missing columns: {missing\_columns}")

# Дополнительные проверки...

return True

def normalize\_email(email):

"""Нормализация email адреса"""

return email.strip().lower()

def normalize\_subject(subject):

"""

Нормализация темы письма

- Удаление RE:, FWD:, номеров в скобках

- Приведение к нижнему регистру

- Удаление лишних пробелов

"""

if not subject:

return ""

# Регулярное выражение для удаления префиксов

pattern = r'^(RE(\[\d+\])?:|FWD?:|\[EXTERNAL\])\s\*'

while True:

new\_subject = re.sub(pattern, '', subject, flags=re.IGNORECASE).strip()

if new\_subject == subject:

break

subject = new\_subject

return subject.lower()

**10. ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ (.env файл)**

Создать .env.example с описанием всех переменных:

# Zoho Mail настройки

ZOHO\_EMAIL=your\_email@zoho.com

ZOHO\_APP\_PASSWORD=your\_app\_specific\_password

# LM Studio настройки

LMSTUDIO\_API\_URL=http://localhost:1234/v1/completions

LM\_MODEL\_NAME=your\_model\_name

# Excel файл (опционально, можно указать путь)

EXCEL\_FILE\_PATH=Mail\_USD.xlsx

# Логирование (опционально)

LOG\_LEVEL=INFO

**11. ТРЕБОВАНИЯ К DEPENDENCIES (requirements.txt)**

pandas>=1.5.0

openpyxl>=3.0.0

requests>=2.28.0

python-dotenv>=0.19.0

PyYAML>=6.0

beautifulsoup4>=4.11.0

email-validator>=1.3.0

**12. СПЕЦИАЛЬНЫЕ ТРЕБОВАНИЯ К ОБРАБОТКЕ**

**Обработка IMAP поиска:**

1. **Экранирование**: Все значения с пробелами, @ символами должны быть в кавычках
2. **Кодировка**: Правильное декодирование заголовков Subject и From
3. **Fallback стратегии**: Если точный поиск не дает результатов, использовать более широкие критерии

**Обработка LM Studio ответов:**

1. **Очистка**: Удаление комментариев // из JSON
2. **Поиск JSON**: Может быть окружен дополнительным текстом
3. **Fallback**: Если парсинг JSON не удался, поместить весь текст в поле "Comments"

**Обработка Excel:**

1. **Резервные копии**: Всегда создавать backup с timestamp
2. **Подсветка**: Измененные ячейки выделять желтым цветом
3. **Валидация**: Проверять структуру файла перед началом работы

**13. ЛОГИРОВАНИЕ СОБЫТИЙ**

Все важные события должны логироваться:

* **INFO**: Начало/завершение процесса, найденные письма, успешные операции
* **WARNING**: Ненайденные ответы, неудачные попытки поиска
* **ERROR**: Ошибки API, проблемы с файлами
* **DEBUG**: Детальная информация о поиске, запросах к API

**14. АРХИТЕКТУРНЫЕ ПРИНЦИПЫ**

1. **Разделение ответственности**: Каждый класс отвечает за свою область
2. **Контекстные менеджеры**: Для автоматического освобождения ресурсов
3. **Обработка ошибок**: На каждом уровне с подробным логированием
4. **Конфигурируемость**: Все настройки в файлах конфигурации
5. **Тестируемость**: Код должен легко покрываться тестами

**15. ФИНАЛЬНЫЕ ТРЕБОВАНИЯ**

При написании кода учитывать:

1. **Производительность**: Не загружать все письма в память одновременно
2. **Безопасность**: Не логировать пароли и чувствительные данные
3. **Читаемость**: Подробные docstrings для всех функций и классов
4. **Совместимость**: Код должен работать на Python 3.8+
5. **Мониторинг**: Детальная статистика выполнения

**🎯 РЕЗУЛЬТАТ**

После выполнения всех требований должно получиться приложение, которое:

* Стабильно работает с большими объемами email
* Корректно обрабатывает различные форматы и кодировки
* Имеет подробное логирование для отладки
* Легко конфигурируется под разные условия
* Обрабатывает ошибки без падения программы
* Предоставляет детальную статистику работы